

Final Report

Title: Last Mouse Lost Game

Swapna Chintapalli
SXC180048

Problem Description

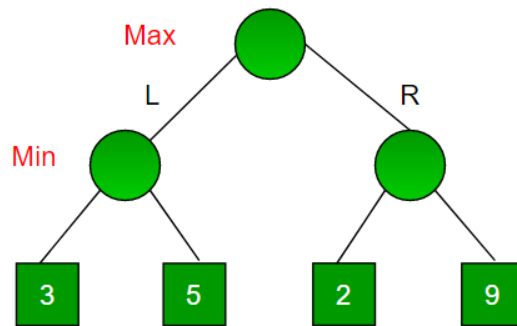
This is a game developed using Artificial Intelligence algorithms. It has a fixed board where there will be set of dots/holes. And the user has to fill the dot to win the game. The main objective of the game is to not select the last dot. You want your opponent to select the last dot on the board in order to win. The game doesn't allow you to make invalid moves like once you fill a dot you cannot undo it. Conditions for the game is that you can select as many dots as you want from one row, but you must select at least one dot and the dots you select must be from the same row. Input a row number and an amount of dots you want to select.

Related Approach:

The first approach followed for this game was to use the minimax algorithm, to calculate the best move by the computer.

Minimax Algorithm:

Minimax (sometimes minmax) is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss while maximizing the potential gain. Alternatively, it can be thought of as maximizing the minimum gain (maximin). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves. It has also been extended to more complex games and to general decision making in the presence of uncertainty. The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, go, chess, and so on. All these games have at least one thing in common, they are logic games. This means that they can be described by a set of rules and premises. With them, it is possible to know from a given point in the game, what are the next available moves. So they also share other characteristic and they are 'full information games'.



Representation of a search tree for a logic game.

There are two players involved, MAX and MIN. A search tree is generated, depth-first, starting with the current game position up to the end game position. Then, the final game position is evaluated from MAX's point of view, as shown in Figure 1. Afterwards, the inner node values of the tree are filled bottom-up with the evaluated values. The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select the minimum value of its children. So what is happening here? The values represent how good a game move is. So the MAX player will try to select the move with highest value in the end. But the MIN player also has something to say about it and he will try to select the moves that are better to him, thus minimizing MAX's outcome.

Optimization

However, only very simple games can have their entire search tree generated in a short time. For most games this isn't possible, the universe would probably vanish first. So there are a few optimizations to add to the algorithm. First a word of caution, optimization comes with a price. When optimizing we are trading the full information about the game's events with probabilities and shortcuts. Instead of knowing the full path that leads to victory, the decisions are made with the path that might lead to victory. If the optimization isn't well chosen, or it is badly applied, then we could end with a dumb AI. And it would have been better to use random moves. One basic optimization is to limit the depth of the search tree. For many games, like chess that have a very big branching factor, this means that the tree might not fit into memory. Even if it did, it would take too long to generate.

The second optimization is to use a function that evaluates the current game position from the point of view of some player. It does this by giving a value to the current state of the game, like counting the number of pieces in the board. Or the number of moves left to the end of the game, or anything else that we might use to give a value to the game position. Instead of evaluating the current game position, the function might calculate how the current game position might help ending the game. Or in another words, how probable is that given the current game position we might win the game. In this case the function is known as an estimation function. This function will have to take into account some heuristics. Heuristics are knowledge that we have about the game, and it can help generate better evaluation functions.

Methods

In order to reduce runtime for this algorithm, following methods/assumptions are made:

1. Symmetry in the board: The board is symmetrical, so making a move on the top row of 2 or the bottom row of two does not make a difference if both rows contain same amount of dots. For example in the first turn making move [0, 2] is equivalent to move [5, 2] (in this example the notation used is [row, amount of dots]). This idea can be taken a step further since the symmetry is not just for rows with the same maximum length. For example if any two rows have 2 dots the moves between those rows are identical. This allowed to cut down the board combinations since you can have many boards that share the same total number of dots in each row but the dots may be in different rows. For example boards were stored in the format of a dictionary where the key is the number of dots and the value is the number of rows with those amount of dots.
2. Counting backwards in board: Since for almost all of the board operations the only thing we care about is how many dots are available, we don't care about the number of x's but only about the number of o's. Therefore instead of having for a loop that counts from 0 to the length of the row, the for loops were changed to count backwards from the length of the row to the first 'x'. This allowed the loop to stop earlier taking much less time when your account for the fact that these functions were being called thousands of times whenever nodes were created.
3. Winning boards: For example the easiest board combination that guarantees you win is {0: 5, 1: 1}. This is a winning board because it will leave only one dot and my opponent must choose the only available dot and I will win. However there are many other more complicated boards you will win if you play the most optimal move, like {0: 3, 1: 1, 2: 1, 3: 1}, {0: 4, 3: 2} and {0: 1, 1: 1, 2: 1, 3: 3} are some examples (the first two are easy to play out and become intuitive, the last one may not seem intuitive as to why it guarantees a win however this is because no matter what move your opponent makes you can get to another winning board in one move).

In the board class within 'board.py' there is a list of winning boards that are organized by how many rows of 0 there are. By using these winning boards you could cut down the tree of this functions calculated moves since once you hit one of these boards you did not have to compute all the moves after since you know that you can always win at this state. This shortened the runtime since you no longer had to calculate the board combinations down to a single dot.

Mirror Strategy:

This strategy as the name implies has the computer mirror the user's move for most of the game. For example if the first move is [0, 2] the computer will decide to move [5, 2] (using the [row, amount of dots] notation). This mirroring occurs for most of the game until one of 2 situations occurs. These situations are shown as 2 if statements within the 'move' function of SmartPlayer.

1. (3 ones + one other row) or (5 ones + one other row) or (1 one + one other row) In this case you want to remove all dots from the row with the most dots, this will result in either the board {0: 3, 1: 3} or {0: 1, 1: 5} or {0: 5, 1: 1} respectively. Each of these boards will have rows each

containing one dot. There is an odd amount of rows which is necessary to ensure the opponent is forced to push the last dot.

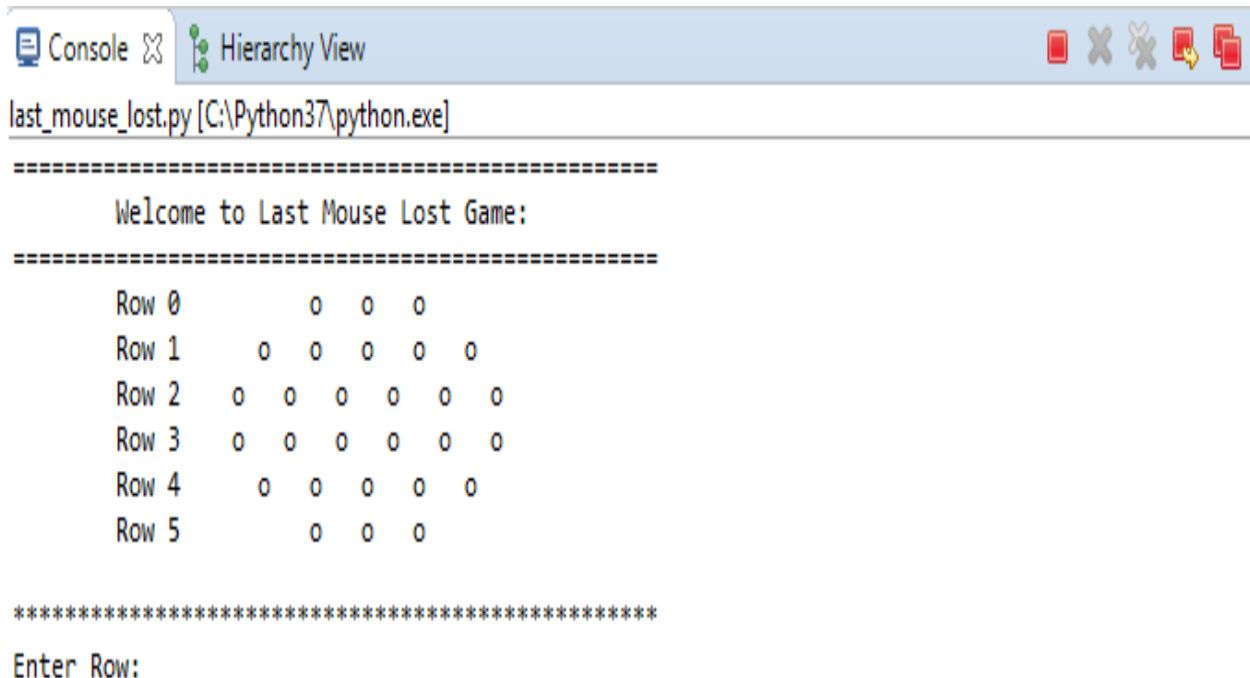
2. (2 ones + one other row) or (4 ones + one other row) or (one row of some amount of dots). This case is very similar to the case above. The only difference is that instead of taking all the dots from the max row, it leaves one dot. This results in the same outcome as in case 1 since you are essentially just adding another row with one dot.

If neither of these cases are fulfilled then the computer will mirror the player until one of these scenarios occur. These scenarios are needed since if they were not in place then the other opponent could just take a row every turn and the computer would delete the last row and lose, or the player could also be a bit smarter and try to outplay the computer however these two cases prevent that. Therefore the Mirror Strategy can be used to guarantee a win if you are the second player in the game.

Screenshots

Players: Human Player and Smart Player (machine)

1. Initial board configuration.



```
last_mouse_lost.py [C:\Python37\python.exe]

=====
Welcome to Last Mouse Lost Game:
=====

Row 0      0  0  0
Row 1     0  0  0  0  0
Row 2    0  0  0  0  0  0
Row 3    0  0  0  0  0  0
Row 4     0  0  0  0  0
Row 5      0  0  0

*****

Enter Row:
```

- Human Player enters the input: row:1 and amount: 3
Smart player makes move (4, 3)

```
Console | Hierarchy View
last_mouse_lost.py [C:\Python37\python.exe]
=====
Welcome to Last Mouse Lost Game:
=====
  Row 0      o o o o
  Row 1      o o o o o
  Row 2      o o o o o o
  Row 3      o o o o o o
  Row 4      o o o o
  Row 5      o o o

=====
Enter Row: 1
Enter Amount: 3
Human Player 0 made move (1, 3)

-----
  Row 0      o o o o
  Row 1      x x x o o
  Row 2      o o o o o o
  Row 3      o o o o o o
  Row 4      o o o o
  Row 5      o o o

=====
Smart Player 1 made move (4, 3)

-----
  Row 0      o o o o
  Row 1      x x x o o
  Row 2      o o o o o o
  Row 3      o o o o o o
  Row 4      x x x o o
  Row 5      o o o

=====
Enter Row:
<
```

- After multiple moves, smart player makes move (3, 2) which leaves only one move to human player with one dot. So that human player has to make that move and will be lost.

```
Console | Hierarchy View
<terminated> last_mouse_lost.py [C:\Python37\python.exe]
=====
Enter Row: 2
Enter Amount: 1
Human Player 0 made move (2, 1)

-----
  Row 0      x x x x
  Row 1      x x x x x
  Row 2      x x x x x o
  Row 3      x x x x o o
  Row 4      x x x x x
  Row 5      x x x

=====
Smart Player 1 made move (3, 2)

-----
  Row 0      x x x x
  Row 1      x x x x x
  Row 2      x x x x x o
  Row 3      x x x x x
  Row 4      x x x x x
  Row 5      x x x

=====
Enter Row: 2
Enter Amount: 1
Human Player 0 made move (2, 1)

-----

Human Player 0 Lost
```

Conclusion

Actual plan for this project is to implement an Artificial Intelligence player that makes reasonable decision by taking moves using Minimax algorithm. The one who selects last dot will lose the game. While implementing, made few assumptions like board symmetry and mirror strategy approach which improved running time for the algorithm.

References

1. Minimax approach to structural optimization problems
<https://link.springer.com/article/10.1007/BF00933350>
2. Walker, P. (published Apr '95). History of Game Theory. Retrieved 11/01/04, from:
<http://william-king.www.drexel.edu/top/class/histf.html>
3. Russell, S. & Norvig, P. Artificial Intelligence: A Modern Approach 3rd Edition, Prentice Hall